International Journal of Advances in Engineering and Management (IJAEM)
Volume 3, Issue 9 Sep 2021, pp: 104-107 www.ijaem.net ISSN: 2395-5252

Design of 16-bit multiplier for Posit data format

Narayana M Hegde[1] and Dr. Kiran V[2]
[1]Student, Department of ECE, RV College of Engineering, Bangalore, Karnataka
[2]Associate Professor, Department of ECE, RV College of Engineering, Bangalore, Karnataka

---

Submitted: 30-08-2021 Revised: 03-09-2021 Accepted: 05-09-2021

ABSTRACT:Multiplication is one of the most commonly used operations of all the arithmetic operations in various applications. In this paper, a multiplier architecture for the posit number system is proposed. Posit number system provides better dynamic range and accuracy compared to floating-point numbers (IEEE 754 standard) for the same word size. The dynamic range and precision of posits can be attributed to an exponent component with run-time varying length. Due to these run-time variations, hardware design is challenging. So, in this paper multiplier for posits is constructed in Verilog HDL.

KEYWORDS: Unum, Posit, Multiplier

I. INTRODUCTION

Posit is a new number system to represent real numbers, proposed by Gustafson as a drop-in replacement for the existing IEEE 754 standard. It is the third revised version of Unum format after type-1 and type-2 unums. It is claimed that posits provide a larger dynamic range and higher accuracy over the same word size. Also due to their tapered decimal accuracy makes them attractive to use in deep learning applications to reduce the number of bits.

Posit number representation is closer to the floating-point standard of representation of real numbers compared to type-1 and type-2 Unum. However, there is an extra field called regime along with the exponent field. The general binary representation format for posits is shown in fig 1. It has four components: sign bit, regime, exponent, and mantissa. The size of regime bits varies at run-time, which makes both exponent and mantissa vary at run time as well. These run-time variations are what make posit provide dynamic range and accuracy.

The value of a number represented in posit format is given by

$$\text{value} = (-1)^{\text{sign bit}} \times \text{useed}^{\text{regime}} \times 2^{\text{exponent}} \times (1 + \text{fraction})$$

where useed = 2^{2es}

The sign bit and regime field are always present in the format. The bit-width of mantissa(including implicit bit) can vary from 1-bit to (N - ES)-bit, where N is the bit width of the posit and ES is the bit-width of the exponent.

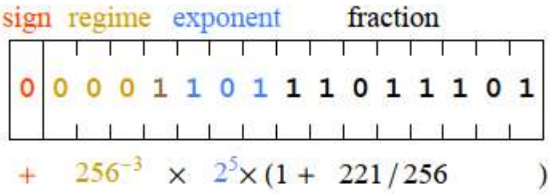


Fig. 1. Posit bit String format

II. MULTIPLIER ARCHITECTURE

In this section, the architecture of the posit multiplier is presented.

The multiplier core consists of three main processing units: posit extraction, multiplier core, and posit packing.

A. Posit data unpacking

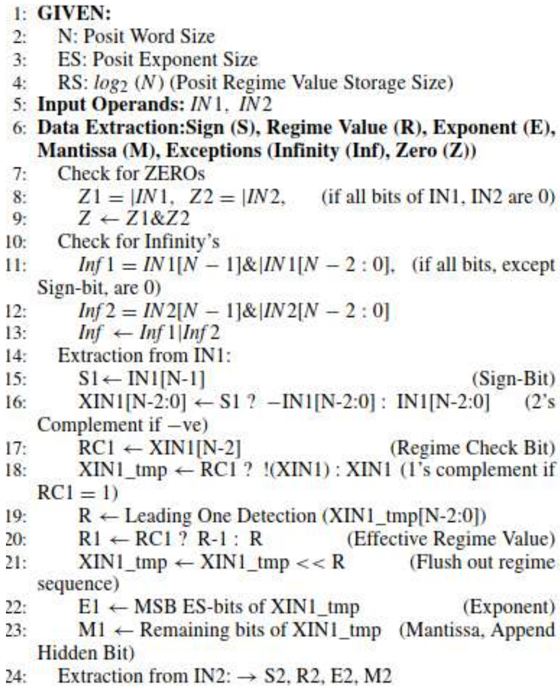
The posit data extraction or unpacking unit takes posit string as input and gives the four fields of posit word as output. First, the operands are checked for exception cases (ZERO and

INFINITY). Posit word with all bits as 0 represents zero and posit string with all bits 0 except the MSB represents infinity

The MSB of the posit word provides the sign of the operand. If the operand is negative MSB is 1 else MSB is 0. For negative operand 2's complement is calculated and the now transformed input operands (without sign bit) are used for further calculations.

The MSB of the transformed operands is used to calculate regime: 0 for the negative regime and 1 for the positive regime. To do this it is required to find the position of terminating regime bit. Leading one detector is used to detect terminating 1 in a sequence of 0s and a leading zero detector is used to detect terminating 0 in a sequence of 1s. This provides the run-time length of the sequence (R). The regime value is R for a sequence of 0 and R-1 for a sequence of 1.

After finding the regime value, the regime sequence is removed by left shifting the operand by R amount using a shifter. Now the exponent and mantissa are aligned at the MSB of the operand. As the exponent length is known, it is usedto extract the exponent from the operand by left-shifting ES times. The remaining bits are mantissa bits. The data extraction algorithm flow is shown in fig 2.



```
GIVEN:
    N: Posit Word Size
    ES: Posit Exponent Size
    RS: $log_2 (N)$ (Posit Regime Value Storage Size)
Input Operands: $IN1, IN2$
Data Extraction:Sign (S), Regime Value (R), Exponent (E), Mantissa (M), Exceptions (Infinity (Inf), Zero (Z))
    Check for ZEROs
        $Z1 = |IN1,\ Z2 = |IN2,$    (if all bits of IN1, IN2 are 0)
        $Z \leftarrow Z1\&Z2$
    Check for Infinity's
        $Inf1 = IN1[N-1]\&|IN1[N-2:0],$   (if all bits, except Sign-bit, are 0)
        $Inf2 = IN2[N-1]\&|IN2[N-2:0]$
        $Inf \leftarrow Inf1|Inf2$
    Extraction from IN1:
        S1 ← IN1[N-1]                                (Sign-Bit)
        XIN1[N-2:0] ← S1 ? −IN1[N-2:0] : IN1[N-2:0]   (2's Complement if −ve)
        RC1 ← XIN1[N-2]                              (Regime Check Bit)
        XIN1_tmp ← RC1 ? !(XIN1) : XIN1   (1's complement if RC1 = 1)
        R ← Leading One Detection (XIN1_tmp[N-2:0])
        R1 ← RC1 ? R-1 : R                           (Effective Regime Value)
        XIN1_tmp ← XIN1_tmp << R                     (Flush out regime sequence)
        E1 ← MSB ES-bits of XIN1_tmp                 (Exponent)
        M1 ← Remaining bits of XIN1_tmp   (Mantissa, Append Hidden Bit)
    Extraction from IN2: → S2, R2, E2, M2
```

Fig 2. Posit data extraction flow [6]

B. Multiplier Core

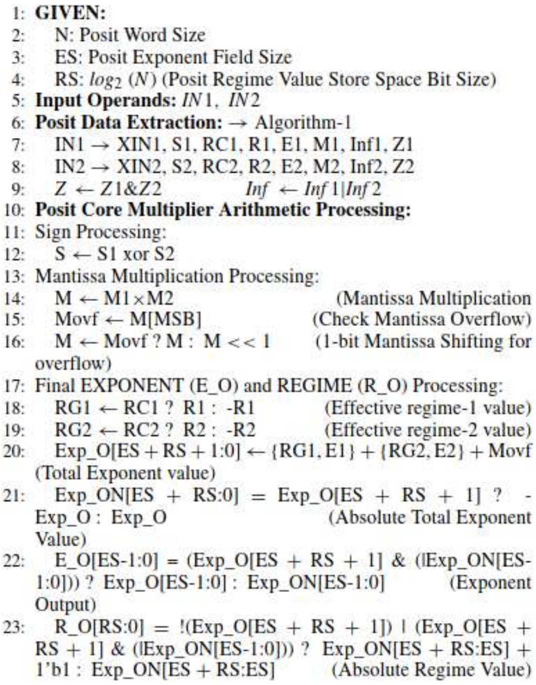
The posit extraction unit provides operands in the form of sign bits (S1, S2), regime check bits (RC1, RC2), regime sequence values (R1, R2), exponent values (E1, E2), mantissas (M1, M2) and infinity and zero checks (INF1, INF2, Z1, Z2).

The sign bit of the product is S1 xor with S2. Mantissas are multiplied using the (N-ES-2) X (N-ES-2) multiplier. The output MSB is checked for multiplication overflow. If there is overflow, the mantissa is left-shifted by 1-bit and the final exponent value is incremented by 1. The regime values R1 and R2 are combined with respective exponent values E1 and E2. The combined values are added along with the overflow bit to obtain the total combined exponent value. The output exponent EO and output regime value RO are calculated from the combined exponent.

The values from the multiplier core are passed to the posit packing unit where posit is constructed.

C. Posit data packing unit

This unit packs the values from the multiplier core into valid posit words. The sign bit is as it is. The regime will consist sequence of 1s for positive exponent and 0s for negative exponent. The same sign bit can be used to terminate the sequence. Next, the exponent value is appended to the regime sequence. The mantissa is rounded and appended at the end. The final posit is the N-bit output of the multiplier. The multiplier flow is shown in fig 3.



```
GIVEN:
    N: Posit Word Size
    ES: Posit Exponent Field Size
    RS: $log_2(N)$ (Posit Regime Value Store Space Bit Size)
Input Operands: $IN1, IN2$
Posit Data Extraction: → Algorithm-1
    IN1 → XIN1, S1, RC1, R1, E1, M1, Inf1, Z1
    IN2 → XIN2, S2, RC2, R2, E2, M2, Inf2, Z2
    $Z \leftarrow Z1\&Z2$          $Inf \leftarrow Inf1|Inf2$
Posit Core Multiplier Arithmetic Processing:
Sign Processing:
    S ← S1 xor S2
Mantissa Multiplication Processing:
    M ← M1×M2                          (Mantissa Multiplication
    Movf ← M[MSB]                      (Check Mantissa Overflow)
    M ← Movf ? M : M << 1              (1-bit Mantissa Shifting for overflow)
Final EXPONENT (E_O) and REGIME (R_O) Processing:
    RG1 ← RC1 ? R1 : -R1               (Effective regime-1 value)
    RG2 ← RC2 ? R2 : -R2               (Effective regime-2 value)
    Exp_O[ES + RS + 1:0] ← {RG1, E1} + {RG2, E2} + Movf (Total Exponent value)
    Exp_ON[ES + RS:0] = Exp_O[ES + RS + 1] ? -Exp_O : Exp_O   (Absolute Total Exponent Value)
    E_O[ES-1:0] = (Exp_O[ES + RS + 1] & (|Exp_ON[ES-1:0])) ? Exp_O[ES-1:0] : Exp_ON[ES-1:0]   (Exponent Output)
    R_O[RS:0] = !(Exp_O[ES + RS + 1]) | (Exp_O[ES + RS + 1] & (|Exp_ON[ES-1:0])) ? Exp_ON[ES + RS:ES] + 1'b1 : Exp_ON[ES + RS:ES]   (Absolute Regime Value)
```

Fig 3. Posit multiplier flow [6]

III. EXPERIMENT AND RESULT

The multiplier for posit is designed for N = 16 and ES = 3 in Verilog HDL. Test vectors to simulate multiplier are generated using python. The test vectors are written to text files in binary format which is then used by the testbench for simulation. The simulator output is shown in fig 4.

Fig 4. Simulation Output

IV. CONCLUSION

In this paper, we designed a multiplier for posit number format with posit bit-size as 16 bits and exponent length as 3. The multiplier is designed in Verilog and simulated using Cadence NC-Verilog Simulator

REFERENCES

[1]. Anderson, M; Tsao, T-C; and Levin, M., 1998, "Adaptive Lift Control for a Camless Electrohydraulic Valvetrain," SAE Paper No. 98102

[2]. Ashhab, M-S; and Stefanopoulou, A., 2000, "Control of a Camless Intake Process – Part II," ASME Journal of Dynamic Systems, Measurement, and Control – March 2000

[3]. Gould, L; Richeson, W; and Erickson, F., 1991, “Performance Evaluation of a Camless Engine Using Valve Actuation with Programmable Timing,” SAE Paper No. 910450.

[4]. Schechter, M.; and Levin, M., 1998, “Camless Engine,” SAE Paper No. 960581

[5]. INTERNATIONAL JOURNAL OF ROBUST AND NONLINEAR CONTROL, Int. J. Robust Nonlinear Control 2001; 11:1023}1042 (DOI: 10.1002/rnc.643).

[6]. M. K. Jaiswal and H. K. -. So, "PACoGen: A Hardware Posit Arithmetic Core Generator," in IEEE Access, vol. 7, pp. 74586-74601, 2019, doi: 10.1109/ACCESS.2019.2920936.